REMARKS/ARGUMENTS

This paper responds to the Office Action of March 4, 2004. Applicant respectfully requests reconsideration of the application.

I. Paragraphs 9-13 of the Office Action

In claims 1 and 22, the use of "device(s)" and "device" is correct. "Device(s)" is used in connection with "memory loads" (plural). "Device" is used with "memory reference" (singular).

In claim 20, the phrase "circuitry in an address translation path" is correct, to indicate that the circuitry in question may be in the circuitry path that performs address translation, even if not in the address translation circuitry itself.

The amendments to the claims in response to paragraph 9 are purely cosmetic, and do not change claim scope. No equivalents are surrendered.

"The instruciton execution circuitry" of claim 57 finds antecedent basis in claim 55, line 6, "instruction execution circuitry ... designed..."

The Office Action objects to the langauge "circuitry and/or software," and states that it is indefintie. The rejection is not understood. "And/or" is a well-established term. For example, Webster's Ninth New Collegiate Dictionary defines "and/or" as follows: "used as a function word to indicate that two words or expressions are to be taken together or individually." "And/or" is not a synonym of "and." If any rejection is maintained, Applicant requests some identification of an ambiguity.

Paragraph 13 of the Office Action is not clear – what is the statutory basis for the rejection? Paragraph 13 uses the form paragraph relating to the "written description" requirement of 35 U.S.C. § 112 ¶ 1, but the reasoning set forth following the form paragraph does not relate to "written description" issues.

In order to advance prosecution, Applicant draws attention to page 37-38 of the specification, which read as follows:

> "Well-behaved memory" is a memory from which a load will receive the data last stored at the memory location. Non-well-behaved memory is typified by memory-mapped device controllers, also called "I/O space," where a read causes the memory to change state, or where a read does not

> necessarily return the value most-recently written, or two successive reads return distinct data.

Applicant also draws attention to the following web pages, excerpts of which appear in Exhibit A to this paper, which show that "side-effect" is a well-established term in the computer arts:

> http://suif.stanford.edu/papers/isca90.pdf
> http://www.cs.caltech.edu/research/ic/transit/tn11/tn11.html

The "side-effects" of an instruction are the changes it makes to the architectural machine state (with certain exceptions, which will be explored if they become relevant).

II. Double patenting

Applicant submits that any double-patenting question is resolved by amendments to the claims.

III. Claims 1, 2, 14, 22 and 30

Initially, Applicant notes that a direct response to paragraph 9 of the Office Action is extraordinarily difficult, in view of the minimal explanation in the Office Action. 37 C.F.R. § 1.104(c)(2) requires that "when a reference is complex ... the particular part relied on must be designated as nearly as practicable." Six repetitions of the phrase "Fig. 2 and associated discussion in the specification related to code morphing software," for unrelated claim elements, and when that software is the subject of the majority of Cmelik's disclosure, is not "particular." Even though the Office Action is too sparse to constitute a *bona fide* rejection, Applicant responds as best possible, and requests that any further examination indicate "particular parts" of references by column and line number, in accordance with Patent Office practice.

Claim 2 recites as follows:

> 2. A method, comprising the step of:
>
> for memory references generated as part of executing a stream of instructions on a computer, evaluating whether an individual memory reference of an instruction references a device having a valid memory address but that cannot be guaranteed to be well-behaved, based at least in part on an annotation encoded in a segment descriptor.

The Office Action is totally silent regarding the "annotation encoded in a segment descriptor," even though this limitation appears in several claims. The closest feature in Cmelik

’992 appears to be the “A/N” bit which Cmelik stores in his TLB (col. 20, line 66 to col. 11, line 1). A “segment descriptor” is not a “TLB,” and thus any § 102 rejection may be withdrawn.

Claims 1, 14, 22 and 30 recite similar language and are patentable for similar reasons.

IV. Claims 1, 40 and 55

Claim 40 recites as follows:

> 40. A method, comprising the steps of:
>
> translating at least a segment of a source program into an object program, the source program instructing a reference execution with a reference sequence of side-effects, the object program instructing an execution in which the sequence of side-effects differs from the reference sequence;
>
> during an execution of the object program on a computer, detecting a side-effect about to be committed to processor state in which the differing side-effect sequence may have a material effect on the execution of the program, and aborting the side-effect;
>
> establishing a program state equivalent to a state that would have occurred in the reference execution; and
>
> resuming execution of the program from the established state in an execution mode that reflects the reference side-effect sequence

Claim 40 recites that certain side-effects are “detected” while still “about to be committed to processor state,” and aborted. In contrast, Cmelik allows analogous side-effects to be “executed.” *E.g.*, col. 17, lines 1-8. The claim then recites “establishing a program state equivalent to a state that would have occurred in the reference execution.” This “establishing” is fairly easy in the context of the claim, where the problematic side-effect has been aborted. It is much more difficult in Cmelik’s scheme, where it has been “executed.”

Claims 1 and 55 recite similar langauge and are patentable for simialr reasons.

V. Dependent claims

Dependent claims 3-13, 15-21, 23-29, 31-39, 41-54, 56-59 and 61-65 are patentable with the independent claims discussed above. In addition, the dependent claims recite additional features that further distinguish the art.

In view of the amendments and remarks, Applicant respectfully submits that the claims are in condition for allowance. Applicant requests that the application be passed to issue in due course. The Examiner is urged to telephone Applicant's undersigned counsel at the number noted below if it will advance the prosecution of this application, or with any suggestion to resolve any condition that would impede allowance. Enclosed is Petition for Extension of Time for two months. In the event that further extension of time is required, Applicant petitions for that extension of time required to make this response timely. Kindly charge any additional fee, or credit any surplus, to Deposit Account No. 23-2405, Order No. 114596-20-4009.

Respectfully submitted,

WILLKIE FARR & GALLAGHER LLP

Dated: August 4, 2004

By: ____________________
David E. Boundy
Registration No. 36,461

WILLKIE FARR & GALLAGHER LLP
787 Seventh Ave.
New York, New York 10019
(212) 728-8000
(212) 728-8111 Fax

Exhibit A to

Response to Office Action

M.I.T.
TRANSIT
PROJECT

Transit Note #11

Early Processor Ideas

Andre DeHon

Tom Knight

Original Issue: May 1990

Last Updated: Tue Nov 9 12:37:36 EST 1993

Acknowledgments: This research is supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

This note documents some early thoughts on a potential processor architecture for Transit. At this point in time, this is mostly a collection of ideas rather than an architectural specification. Ideas and counterproposal are welcomed.

Goals

- High Scalar Performance
- Efficient Symbolic Processing
- Efficient execution of popular programming languages and paradigms (*e.g.* C, C++, FORTRAN)
- Support for multiple contexts with inexpensive context switching
- Support for fast procedure calls and returns
- Implicit support for use in large, non-bus oriented multiprocessor computer.
- *Graceful Degradation / Fault-Tolerance ?*

Basic Organization

Summary

To summarize, the goals of this memory interface are as follows:

1. Decrease the frequency of long-latency operations by warning the memory system about data requirements in advance of the immediate need

2. Allow the processor to perform useful computations while waiting for memory operations to complete without incurring costly overhead to perform the computations or return to the original instruction stream

3. Provide a mechanism for timely resumption of instructions streams when memory is prepared to complete pending operations

I-Cache

The I-cache is an on chip cache for the instruction stream. This will probably be a direct mapped cache. The size will depend on available space on the chip.

[The I-cache may want to be organized to hold branch targets to minimize the performance degredation when not executing sequential instructions. The sequential instruction fetches can be pipelined. If this is done properly, sequential execution can be satisfied by pipelined memory fetches. No processor stalls occur unless a target of a branch is not in the I-cache. -- TK]

Register Loading/Unloading

From time to time it will be appropriate to change the contexts/processes which are resident on the processor. Unloading a context will require storing all the state associate with a given context out to main memory. The state of a context will be its general registers, condition registers, and any special registers (**PC**, **FP**, *etc.*). If the pipeline is organized so that instruction do not side-effect processor state until the final pipeline stage, it will only be necessary to clean out the pipeline for the next loaded processes. Otherwise, it may be necessary to allow the instructions in the pipeline to clear before unloading the context. Loading a new context will essentially require loading the same set of information.

We really do not want to be forced to complete the execution of instructions in the pipeline. We have no way of guaranteeing that the instructions in the pipeline can complete in a short amount of time. Often it is the case that the context is being unloaded because it has encountered an extremely long latency operation. So we really want to be able to simply flush the pipeline and restart at the appropriate address. For this to be possible, it must be the case that processor state is only side-effected in the final pipeline stage.

One question to consider, is how loading and unloading gets done. It might be possible to have a separate unit unloading and reloading contexts in parallel with processor execution. However, this might require too many resources to be practical (*e.g.* separate read/write port into the register file, shared access to load/store path (or separate load/store path), access to registers not in current context, non-trivial control unit). Alternately, register loading and unloading can be done explicitly.

Boosting Beyond Static Scheduling in a Superscalar Processor

Michael D. Smith, Monica S. Lam, and Mark A. Horowitz
Computer Systems Laboratory
Stanford University, Stanford CA 94305-4070

May 1990

1 Introduction

Superscalar processors are uniprocessor organizations capable of increasing machine performance by executing multiple scalar instructions in parallel. Since the amount of instruction-level parallelism within a basic block is small, superscalar processors must look across basic block boundaries to increase performance. In numerical code, the do-loop branches, which are a large percentage of the total branches executed, can be resolved early to expose the parallelism between many iterations. Unfortunately, many of the branches in non-numerical code are data dependent and cannot be resolved early. Thus, speculative execution, the execution of operations before previous branches, is an important source of parallelism in this type of code.

Instruction-level parallelism can be extracted statically (at compile-time) or dynamically (at run-time). Statically-scheduled superscalar processors and Very Long Instruction Word (VLIW) machines exploit instruction-level parallelism with a modest amount of hardware by exposing the machine's parallel architecture in the instruction set. For numerical applications, where branches can be determined early, compilers harness the parallelism across basic blocks by techniques such as software pipelining [11] or trace scheduling [7]. However, the overhead and complexity of speculative computation in compilers has prevented efficient parallelization of non-numerical code.

Dynamically-scheduled superscalar processors, on the other hand, effectively support speculative execution in hardware. By using simple buffers, these processors can efficiently commit or discard the side effects of speculative computations. Unfortunately, the additional hardware necessary to look far ahead in the dynamic instruction stream, find independent operations, and schedule these independent operations out of order is costly and complex.

We are interested in using superscalar techniques to increase the performance of non-numerical code at a reasonable cost. To accomplish this goal, we propose a superscalar architecture, which we call *TORCH*, that combines the strengths of static and dynamic instruction scheduling. The strength of static scheduling is the compiler's ability to efficiently schedule operations across many basic blocks; consequently, TORCH performs all instruction scheduling in the compiler. The strength of dynamic scheduling is in its ability to efficiently support speculative execution; consequently, TORCH provides hardware that allows the compiler to schedule any instruction before preceding branches, an operation we term *boosting*. Boosted instructions are conditionally committed upon the result of later branch instructions. Boosting, therefore, removes the scheduling constraints that result from the dependences caused by conditional branches and makes aggressive instruction scheduling in the compiler simple.

To make the conditional evaluation of boosted instructions efficient at run-time, the TORCH hardware includes two shadow structures: the *shadow register file* and *shadow store buffer*. These structures buffer the side effects of a boosted instruction until its dependent branch conditions are determined. On a correctly predicted branch, the hardware commits the appropriate values in the shadow structures. On a mispredicted branch, the hardware guarantees correct program operation by squashing all shadow values.

In this paper, we overview the TORCH architecture, describe the TORCH hardware to support boosting, and present the results of a simple static scheduler which performed limited instruction boosting and no load/store reorganization. The simple scheduler and our evaluation system allow us to quickly assess the viability of are issued and executed in parallel. There is no overhead during run-time to schedule instructions, and the hardware is simple. The compiler essentially has a infinite instruction window and uses global program knowledge, program semantics (dependences), and resource constraints in constructing each instruction schedule. Finally, instruction alignment is not an issue in static scheduling since the compiler schedules instructions in fetch blocks.

The basis for statically-scheduled superscalar processors comes from the field of horizontally-microcoded machines and Very Long Instruction Word (VLIW) architectures. Early on, VLIW machines were similar to horizontally-coded microcode in that each instruction word had a field for each independent functional unit [3]. More recent VLIW machines [5] remove this restriction by providing dynamic NOPs for idle functional units. In fact, the distinction between statically-scheduled superscalar processors and VLIW machines is blurry. Both machines rely on the compiler to explicitly specify the instruction-level parallelism and manage the hardware resources. The difference between statically-scheduled superscalar processors and VLIW machines is basically one of terminology. VLIW machines refer to operations within a singly-fetched instruction, while statically-scheduled superscalar processors refer to instructions within a single fetch block. A VLIW operation is equivalent to a superscalar instruction since both control a single functional unit. Many statically-scheduled machines have been announced either as superscalar processors [1, 17] or as VLIW processors [4].

Unfortunately, all these machines encounter difficulties when scheduling across conditional branches even though software branch prediction can be as accurate as hardware branch prediction [13]. Delay-branch schedulers are able to perform some limited movement of instructions across basic block boundaries. The compiler either moves instructions down from within a basic block to fill the branch delay slot; the branch must not depend upon these instructions so that it does not matter whether the machine executes them before or after the branch. The other way to fill branch delay slots is to lift an instruction up from the fall-through or target basic block. These instructions must not have any side effects so that their execution does not affect the machine state if the branch goes the other direction. Though these schemes are able to effectively fill a single branch slot, the effectiveness of these motions drops off significantly as the number of branch slots increases. For instance, a compiler can fill approximately 70% of single branch slots, but only 25% of double branch slots [13]. As we can see from the percentages of multiple branch slots filled, these reorganizing schemes are very limited in their ability to move instructions across conditional branches.

Trace scheduling [7] is a compiler technique that was designed to increase the compiler's ability to move instructions across basic block boundaries. It utilizes branch predictions to create a single long block of code out of individual basic blocks without hardware support. A greater level of concurrency is achieved by scheduling this long block instead of the individual basic blocks, but at the expense of fix-up code at the entry and exit points of the trace. The difficulty and overhead of saving and restoring state for instructions with side effects and of finding a few major traces in non-numerical code with its large number of run-time branches makes trace scheduling an unattractive choice.

Other statically-scheduled machines reduce the effects of conditional branches by scheduling both paths of a conditional branch in parallel [18]. This scheme requires hardware support to NOP the instructions along the non-taken branch path. Though this technique allows for some overlap between the schedules of the basic blocks before and after the conditional statement with the combined branches of the conditional statement, the compiler's ability to move instructions with side-effects is not improved.

2.3 Scheduling in TORCH

Boosting combines into one architecture the best aspects of static and dynamic scheduling to overcome the difficulties of scheduling instructions across conditional branches. Boosting relies on the compiler's knowledge of the program semantics and ability to explore many schedules to find the best instruction schedule. Boosting simplifies scheduling by assigning the hardware the responsibility of handling side effects. To the scheduler, all boosted instructions appear free from side effects.

The hardware efficiently handles boosting by providing extra buffering in the register file and store buffer. The shadow structures hold the results of boosted instructions until they are committed or squashed. Efficiency is guaranteed by performing the commit and squash operations without any performance penalty. For further efficiency, the hardware postpones all exception processing on boosted instructions until the hardware tries to commit the boosted instructions. Exceptions are precise and easy to identify.